

Software Engineering Education: On the Right Track with Ada

John W. McCormick
Computer Science Department
University of Northern Iowa
Cedar Falls, IA 50614-0507
mccormick@cs.uni.edu

A version of this article previously appeared in CrossTalk 13:8, (August 2000), 22-25.

Abstract

An introductory undergraduate course in real-time embedded software development should acquaint students with the fundamental scientific issues of real-time computing and practical skills in software development. While the theoretical issues can be covered without a laboratory, real-time embedded programming skills require the experiences that a laboratory provides. A major problem is finding equipment suitable for teaching these skills. For over a decade I have used a computer controlled model railroad in my real-time embedded systems course

Introduction

Nearly all modern devices contain embedded software. Even a few years ago, the typical new car from General Motors contained \$675 of steel and nearly \$2,500 of electronics including a dozen or so embedded microprocessors. Often, the software in these embedded systems must execute in real-time for the equipment to function correctly. Despite the need for skilled real-time embedded software developers, there is little attention paid to this area of software development in the undergraduate computer science curriculum.

An introductory undergraduate course in real-time embedded software development should acquaint students with the fundamental scientific issues of real-time computing and practical skills in software development. While the theoretical issues can be covered without a laboratory, real-time embedded programming skills require the experiences that a laboratory provides. A major problem is finding equipment suitable for teaching these skills.

Simulators are commonly used to give students experience with real-time programming. Typically these simulators do not provide many of the frustrating problems associated with physical systems. Hardware and software development are parallel activities in many embedded systems projects. Gathering evidence for the determination of whether a fault is in the hardware or the software is an important skill for the embedded systems programmer. Lack of experience with real systems is one reason cited by engineers who would exclude computer science graduates from their development teams.

For over a decade I have used a computer controlled model railroad in my real-time embedded systems course. Some of the advantages of using a model railroad in the laboratory include

- Model railroad equipment is readily available and priced well below typical laboratory equipment.
- Model railroads provide a wealth of problems from both the discrete and continuous real-time domains.
- The electronics are easily understood by most undergraduate computer science students.
- Students are highly enthusiastic about writing software to control a model train layout.

As a direct result of presentation and publication of previous work [1, 2, 3, 4], over 50 organizations have requested detailed specifications of the laboratory. All but three were discouraged by the amount of effort (over 500 hours) required to assemble the necessary interface electronics. With the support of the Maytag and Rockwell Foundations, I am now implementing an *affordable* real-time embedded systems laboratory that other institutions can *easily* duplicate.

The Real-Time Systems Course

The computer science curriculum at the State University of New York (SUNY) at Plattsburgh includes a specialized track called Computer Controlled Systems. This track was developed for students interested in the specification, design, and implementation of real-time embedded software. In addition to the typical courses in a computer science curriculum, this track includes more courses in continuous mathematics, physics, and electronics. The departments of Computer Science and Industrial Technology at the University of Northern Iowa (UNI) are currently designing a joint Computer Controlled Systems curriculum.

The real-time systems course serves as the capstone course of the curriculum. To perform well in this course, students must integrate knowledge from their previous work in computer science, electronics, English, mathematics, and physics. Students are exposed to the fundamental scientific issues in real-time computing and gain practical skills of software development. A major goal is to train software engineers capable of working as part of an interdisciplinary development team. Many topics are covered at a survey level. For example, students in the course learn just enough of the basic concepts of control theory to be able to communicate with a control engineer and to implement a simple PID control algorithm. Feedback from employers in a wide range of domains including avionics, communications, manufacturing, and medical instrumentation has been extraordinarily positive.

Laboratory Assignments

The four credit hour course has three fifty minute "lectures" and a three hour laboratory session each week. The early laboratory sessions are used to review (or learn) and practice with the features of the implementation language that are important for the completion of their project. These include data modeling, encapsulation and reuse, concurrent programming, and exceptions. Later laboratory sessions are devoted to developing code that will be directly applied to their projects including

- Polling and interrupt based device drivers
- Implementation of a whistle class
- Implementation of a turnout class

Turnouts are electro-mechanical devices which sometimes fail to operate correctly. The software must detect and correct turnout failures. Students derive their code from state machines they develop in one of the lecture sessions.

Course Project

Students are divided into teams of three or four students to complete a substantial (12-15K lines) project. Teams are free to formulate their own projects. Minimum project requirements include

- Running multiple trains
- At least one train controlled by a human engineer
- No collisions
- Detection and recovery from hardware failures including
 - Turnouts
 - Sensors
 - Lost cars
 - Devious professors

Over the years, train races, train wars, and scheduling problems have been the most popular project themes. Deliverables for the project include:

- System concept document
- Detailed user's manual
- OMT documents
 - Object model diagrams
 - Dynamic model diagrams
 - Functional model diagrams
 - Data dictionary
- Compiled class specifications
- Unit (class) test plans

These deliverables are used as milestones throughout the course to help ensure that students keep up with the demanding schedule necessary to complete the project. One of my major tasks is to work with teams on their systems concept document to reduce overly optimistic proposals into ones that can be completed. Students are aware of the completion rates of past teams (presented later in this paper) so they understand that they can complete the project by the end of the semester.

Student teams do fairly exhaustive module testing where the behavior of a particular object (say a turnout or locomotive) is well understood. Integration testing is bounded by the end of the semester.

The Laboratory

My first model railroad laboratory was constructed in 1983 at SUNY Plattsburgh. Construction of a new railroad layout at the University of Northern Iowa is currently underway.

Railroad Hardware

The model railroads are HO scale. While smaller scales would permit more equipment in the laboratory, they are more expensive, more difficult to maintain, and less readily available.

To run multiple trains on their layouts, model railroaders traditionally divide the track into electrically isolated sections called blocks. Many toggle and rotary switches are used to connect a particular power supply (called a cab) to a group of track blocks beneath each train. In our layout, the computer controls the voltage and polarity applied to each of the blocks. Our current UNI layout design has 40 blocks. Today's model railroad enthusiasts often use more modern direct digital control of locomotives to solve the problem of multiple train control. We have rejected this approach as it provides fewer software development problems to our students and less experience with analog electronics.

Turnouts are controlled by gear and screw driven switch machines. The computer can determine and modify the state of each turnout. Our current UNI layout design has 26 turnouts.

In order to do closed loop control, it is necessary to obtain feedback on the process being controlled. For the model train this feedback consists of the trains' locations as a function of time. This information is obtained in two ways:

1. Hall effect sensors installed on the track. These are triggered by small magnets attached to the front of every locomotive and to the rear of each caboose.
2. A radio link installed in a box car that sends a pulse with every wheel rotation.

Our current UNI layout design has 55 Hall effect sensors. The radio link allows us to determine a train's position information to within about 1 cm. From the data obtained from the link we can also calculate the train's speed. Currently there are two problems associated with the radio link. The wheels on the box car slip on the track as the car moves. Thus the calculated distance moved by the train is less than the actual distance. This error increases with time. This problem is a *good* problem as students can easily correct for the slippage by using the positions obtained from the fixed sensors. The second problem is a result of recycling radio transmitters from very inexpensive toys. The transmitters broadcast over a large portion of the frequency spectrum making it impossible to use multiple transmitters at the same time. We are currently working on a design to replace the radios with an IR link.

A final piece of railroad hardware is a hand-held control cab. This is a small box with buttons, knobs, and toggle switches that a human engineer can use to control a train. Typical student

projects assign knobs for train throttles, buttons for whistles and brakes, and toggle switches for train direction (forward or reverse) and for setting the next turnout ahead of the train (left or right).

Computing Hardware

A number of different hardware configurations have been used over the long history of this project. In our first laboratory, students developed their control software on a Digital Equipment Corporation PDP 11/24. They used a serial link to download their executable programs to a PDP 11/23 computer. In 1989 I received a laboratory improvement grant from the NSF. This grant enabled me to replace the PDP 11/24 with a microVAX II and the PDP 11/23 with an rtVAX (a microVAX computer optimized for real-time). The system currently under design at UNI uses PCs for software development. Two or three inexpensive networked microcomputers will boot and execute the software developed by the students.

Interface Hardware

The interface hardware connects the control computers to the railroad hardware. Figure 1 is a diagram showing the layers in the system. One or more CPUs are connected to commercially available analog to digital converters (ADC), digital to analog converters (DAC), TTL level digital I/O (DIO), and counter/timers. The connection may be made through any of a number of different buses such as ISA, EISA, PCI, GPIB, CAN, USB, and even standard serial or parallel ports. We use custom hardware to connect these devices to the railroad layout. In the past, this interface layer was hand built on wire wrapped and soldered prototyping boards. It took considerable effort to construct it. With the support of the Maytag Foundation and Rockwell, we are now in the process of designing and manufacturing circuit boards that will make this aspect of building the laboratory much easier for us and other schools that wish to duplicate our efforts. The interface hardware consists of three subsystems (block control, turnout control, and train sensors) detailed in the following sections.

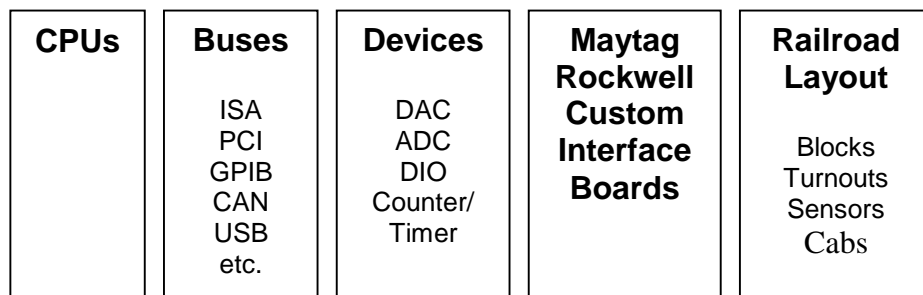


Figure 1 Hardware Layers Connecting the Control Computers to the Model Railroad

Block Control

The block control subsystem controls the voltage and polarity applied to each track block in the railroad layout. Figure 2 shows a single track block circuit.

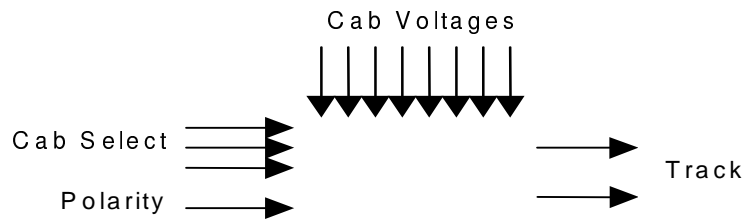


Figure 2 Track Block Control Circuit

The two analog outputs are connected to the rails of a track block to supply power to the train on that block. Each circuit has four digital inputs and eight analog inputs. Three of the digital inputs (Cab Select) are used to select which of the eight analog inputs will be used to power the track block. The remaining digital input is used to select the polarity of the voltage applied to the track. The analog inputs (marked Cab Voltages in Figure 2) may be supplied by digital to analog converters or by programmable counter/timers. The latter uses pulse width modulation to control the speed of a train. We expect that each of our block control boards will contain six or twelve block control circuits.

Turnout Control

This circuit controls a Tortoise brand switch machine. These switch machines take three to five seconds to change the direction of a turnout. There are four possible states for a turnout: left, right, moving left, and moving right. Each circuit uses one output bit to set the direction of the turnout. Rather than use two input bits to determine the state of the switch machine we use the output bit in combination with one input bit that reports whether the turnout has reached the desired direction.

Train Sensors

This circuit connects the Hall effect sensors on the track to a DIO board with interrupt capabilities. We place these sensors on the boundaries between track blocks. When a locomotive is detected, the software must power up the next block before the wheels bridge the gap between blocks. This is a hard real-time deadline in the system as the block power supply fuse will blow if the software fails to power the next block in time .

Software

During the first six years that the real-time systems course was offered, students developed their control code in C. As shown in Figure 3, no team successfully implemented the minimum project requirements when the C language was used. To ease student and teacher frustrations I made an increasing amount of my solutions available to the teams. Figure 3 shows that even when I provided nearly 60% of the project code, no team was successful in implementing the minimum requirements.

Along with the new hardware provided by the NSF funding for the project was a collection of DEC compilers. Thinking that the low level of tasking provided through semaphores was the major contributor to the problem I selected a language with a much higher level of tasking abstractions — Ada. I expected a disaster the first year with the new equipment and new language. As in a real-life embedded systems project, I was building the hardware while my students were writing the software. I finished the hardware with only 4 weeks remaining in the semester. But to my amazement, nearly fifty percent of the student teams had their projects working before the end of the semester. I had only supplied them with two sample device drivers. As shown in Figure 4, when I supplied some additional software components (simple window packages not relevant to the real-time aspect of the project), over 75% of my teams routinely completed their projects.

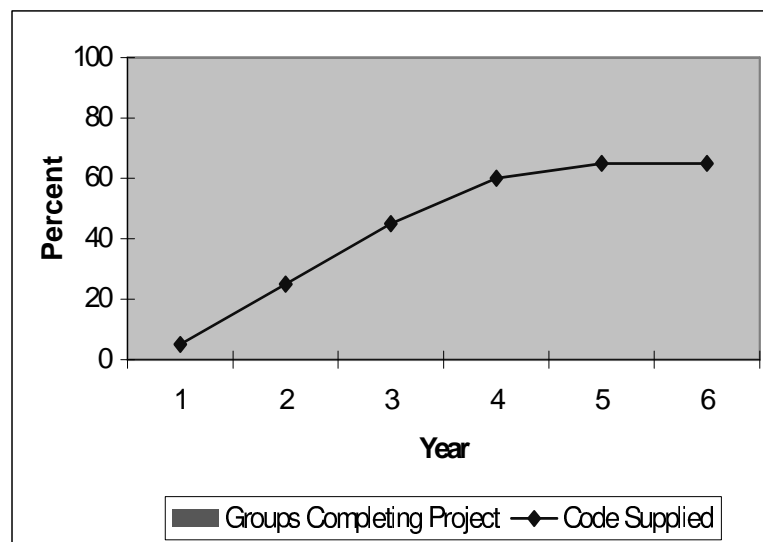


Figure 3 C Language: Completion Rate (zero) and Amount of Code Supplied

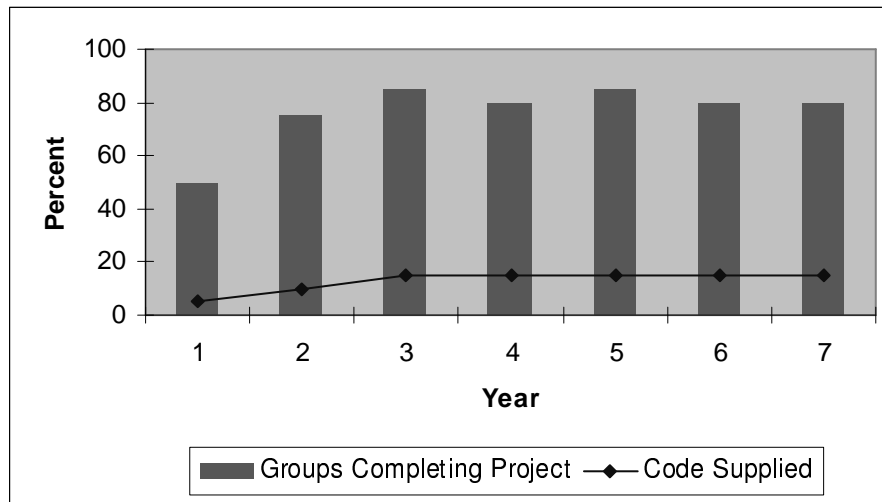


Figure 4 Ada Language: Completion Rate and Amount of Code Supplied

Why Does Ada Succeed where C Fails?

The only difference between the years in which teams succeeded in implementing their projects and those in which no team succeeded was the implementation language. The project specification, design, and unit testing techniques did not vary. While the new computing hardware used by the Ada teams was more modern (faster and fewer breakdowns) it provided no significant implementation advantages. Upon reading the project listings and team member diaries I conclude that the major advantages of Ada for these students were (in order of importance)

- Modeling of scalar objects
 - Strong typing
 - Range constraints
 - Enumeration types
- Parameter modes that reflect the problem rather than the mechanism
- Named parameter association
- Arrays whose indices do not have to begin at zero
- Representation clauses for device registers (record field selection rather than bit masks)
- Higher level of abstraction for tasking (rendezvous rather than semaphores)
- Exception handling
- A compilation model that detects obsolete units

I found that my original hypothesis that C's low level tasking mechanism was the major problem to be incorrect. While Ada's high level of abstraction for tasking was helpful to the students, it was the accurate modeling of scalar quantities that contributed the most to Ada's success in this course. This is consistent with studies done on the nature of wicked bugs in software [5] where

nearly 80% of programming errors in the C/C++ programs studied were a result of problems with scalars.

Conclusions

The model railroad provides an exciting environment for teaching a course in real-time embedded systems. With the support the Maytag Foundation and Rockwell we are developing the interface hardware to allow us and other schools to easily connect a variety of computers to a model railroad at minimal cost. UNI will make the interface boards we design and manufacture available to all. Contact me for details.

References

1. McCormick, J.W. (1988). Using a Model Railroad to Teach Digital Process Control. SIGCSE Bulletin, 20, 304-308.
2. McCormick, J.W. (1991). A Laboratory for Teaching the Development of Real-Time Software Systems. SIGCSE Bulletin, 23, 260-264.
3. McCormick, J.W. (1992). A Model Railroad for Ada and Software Engineering. Communications of the ACM, 35, 68-70.
4. McCormick, J.W., Kudrle, J., & Poulin, J.M. (1994) Ada, Objects, and Model Trains. The Proceedings of the Eighth Annual Software Engineering Education and Training Symposium, Albuquerque, NM, 8, 29-33.
5. Eisenstadt, M. (1997). My Hairiest Bug War Stories. Communications of the ACM, 40, 30-37.